kwplot Documentation

Release 0.4.12

Jon Crall

Jun 16, 2022

CONTENTS

CHAPTER

ONE

KWPLOT - THE KITWARE PLOT MODULE

This module is a small wrapper around matplotlib and seaborn that simplifies develoepr workflow when working with code that might be run in IPython or in a script. This is primarilly handled by the *kwplot.auto_backends* module, which exposes the functions: *kwplot.autompl()*, *kwplot.autoplt()*, and *kwplot.autosns()* for auto-initialization of matplotlib, pyplot, and seaborn.

A very common anti-pattern in developer code is importing `matplotlib.pyplot` at the top level of your module. This is a mistake because importing pyplot has side-effects which can cause problems if executed at a module level (i.e. they happen at import time! Anyone using your library will have to deal with these consequences)

To mitigate this we recommend only using pyplot inside of the scope of the functions that need it.

Importing *kwplot* itself has no import-time side effects, so it is safe to put it as a module level import, however, plotting is often an optional feature of any library, so we still recomend putting that code inside the functions that need it.

The general code flow looks like this, inside your function run:

```
import kwplot
kwplot.autompl()

# Pyplot is now initialized do matplotlib or pyplot stuff
...
```

This checks if you are running interactively in IPython, if so try to use a Qt backend. If not, then try to use a headless Agg backend.

You can also do

```
import kwplot
# These also call autompl in the backend, and return either the seaborn or
# pyplot modules, so you dont have to import them in your code. When
# running seaborn, this will also call ``sns.set()`` for you.
sns = kwplot.autosns()
plt = kwplot.autoplt()
...
```

In addition to this auto-backend feature, kwplot also exposes useful helper methods for common drawing operations.

There is also a small CLI that can be used to view multispectral or uint16 images.

CHAPTER

TWO

KWPLOT PACKAGE

2.1 Submodules

2.1.1 kwplot.auto_backends module

This module handles automatically determening a "good" matplotlib backend to use before importing pyplot.

kwplot.auto_backends.**autompl**(*verbose=0*, *recheck=False*, *force=None*)

Uses platform heuristics to automatically set the matplotlib backend. If no display is available it will be set to *agg*, otherwise we will try to use the cross-platform *Qt5Agg* backend.

Parameters

- **verbose** (*int, default=0*) – verbosity level
- **recheck** (*bool, default=False*) – if False, this function will not run if it has already been called (this can save a significant amount of time).
- **force** (*str, default=None*) – backend to force to or "auto"

CommandLine

```
# Checks
export QT_DEBUG_PLUGINS=1
xdoctest -m kwplot.auto_backends autompl --check
KWPLOT_UNSAFE=1 xdoctest -m kwplot.auto_backends autompl --check
KWPLOT_UNSAFE=0 xdoctest -m kwplot.auto_backends autompl --check
```

Example

```
>>> # xdoctest +REQUIRES(--check)
>>> plt = autoplt(verbose=1)
>>> plt.figure()
```

References

https://stackoverflow.com/questions/637005/check-if-x-server-is-running

kwplot.auto_backends.**autoplt**(*verbose=0*, *recheck=False*, *force=None*)

Like autompl, but also returns the `matplotlib.pyplot` module for convenience.

Returns
ModuleType

kwplot.auto_backends.**autosns**(*verbose=0*, *recheck=False*, *force=None*)

Like autompl, but also calls `seaborn.set()` and returns the `seaborn` module for convenience.

Returns
ModuleType

kwplot.auto_backends.**set_mpl_backend**(*backend*, *verbose=None*)

Parameters
backend (*str*) – name of backend to use (e.g. Agg, PyQt)

class kwplot.auto_backends.**BackendContext**(*backend*)

Bases: `object`

Context manager that ensures a specific backend, but then reverts after the context has ended.

Because this changes the backend after pyplot has initialized, there is a chance for odd behavior to occur. Please submit and issue if you experience this and can document the environment that caused it.

CommandLine

```
# Checks
xdoctest -m kwplot.auto_backends BackendContext --check
```

Example

```
>>> # xdoctest +REQUIRES(--check)
>>> from kwplot.auto_backends import *  # NOQA
>>> import matplotlib as mpl
>>> import kwplot
>>> print(mpl.get_backend())
>>> #kwplot.autompl(force='auto')
>>> #print(mpl.get_backend())
>>> #fig1 = kwplot.figure(fnum=3)
>>> #print(mpl.get_backend())
>>> with BackendContext('agg'):
>>>     print(mpl.get_backend())
>>>     fig2 = kwplot.figure(fnum=4)
>>> print(mpl.get_backend())
```

2.1.2 kwplot.draw_conv module

Helper for drawing convolutional neural network weights.

This may be removed in the future.

kwplot.draw_conv.**make_conv_images**(*conv*, *color=None*, *norm_per_feat=True*)

Convert convolutional weights to a list of visualize-able images

Parameters

- **conv** (*torch.nn.Conv2d*) – a torch convolutional layer
- **color** (*bool*) – if True output images are colorized
- **norm_per_feat** (*bool*) – if True normalizes over each feature separately, otherwise normalizes all features together.

Return type
ndarray

Todo:

- [ ] better normalization options

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> conv = torch.nn.Conv2d(3, 9, (5, 7))
>>> weights_tohack = conv.weight[0:7].data.numpy()
>>> weights_flat = make_conv_images(conv, norm_per_feat=False)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwimage
>>> import kwplot
>>> stacked = kwimage.stack_images_grid(weights_flat, chunksize=5, overlap=-1)
>>> kwplot.imshow(stacked)
>>> kwplot.show_if_requested()
```

kwplot.draw_conv.**plot_convolutional_features**(*conv*, *limit=144*, *colorspace='rgb'*, *fnum=None*, *nCols=None*, *voxels=False*, *alpha=0.2*, *labels=False*, *normaxis=None*, *_hack_2drows=False*)

Plots the convolutional layers to a matplotlib pyplot.

The convolutional filters (kernels) are stored into a grid and saved to disk as a Maplotlib figure. The convolutional filters, if it has one channel, will be stored as an intensity imgage. If a colorspace is specified and there are three input channels, the convolutional filters will be represented as an RGB image.

In the event that 2 or 4+ filters are displayed, the different channels will be flattened and showed as distinct outputs in the grid.

Todo:

- [ ] refactor to use make_conv_images

Parameters

- **conv** (*torch.nn.modules.conv._ConvNd*) – torch convolutional layer with weights to draw
- **limit** (*int*) – the limit on the number of filters drawn in the figure, achieved by simply dropping any filters past the limit starting at the first filter. Detaults to 144.
- **colorspace** (*str*) – the colorspace seen by the convolutional filter (if applicable), so we can convert to rgb for display.
- **voxels** (*bool*) – if True, and we have a 3d conv, show the voxels
- **alpha** (*float*) – only applicable if voxels=True
- **stride** (*list*) – only applicable if voxels=True

Returns
fig - a Matplotlib figure

Return type
matplotlib.figure.Figure

References

https://matplotlib.org/devdocs/gallery/mplot3d/voxels.html

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> conv = torch.nn.Conv2d(3, 9, (5, 7))
>>> plot_convolutional_features(conv, colorspace=None, fnum=None, limit=2)
```

Example

```
>>> # xdoctest: +REQUIRES(--comprehensive)
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torchvision
>>> # 2d uncolored gray-images
>>> conv = torch.nn.Conv3d(1, 2, (3, 4, 5))
>>> plot_convolutional_features(conv, colorspace=None, fnum=1, limit=2)
```

```
>>> # 2d colored rgb-images
>>> conv = torch.nn.Conv3d(3, 2, (6, 4, 5))
>>> plot_convolutional_features(conv, colorspace='rgb', fnum=1, limit=2)
```

```
>>> # 2d uncolored rgb-images
>>> conv = torch.nn.Conv3d(3, 2, (6, 4, 5))
>>> plot_convolutional_features(conv, colorspace=None, fnum=1, limit=2)
```

```
>>> # 3d gray voxels
>>> conv = torch.nn.Conv3d(1, 2, (6, 4, 5))
>>> plot_convolutional_features(conv, colorspace=None, fnum=1, voxels=True,
>>>                             limit=2)
```

```
>>> # 3d color voxels
>>> conv = torch.nn.Conv3d(3, 2, (6, 4, 5))
>>> plot_convolutional_features(conv, colorspace='rgb', fnum=1,
>>>                             voxels=True, alpha=1, limit=3)
```

```
>>> # hack the nice resnet weights into 3d-space
>>> # xdoctest: +REQUIRES(--network)
>>> import torchvision
>>> model = torchvision.models.resnet50(pretrained=True)
>>> conv = torch.nn.Conv3d(3, 1, (7, 7, 7))
>>> weights_tohack = model.conv1.weight[0:7].data.numpy()
>>> # normalize each weight for nice colors, then place in the conv3d
>>> for w in weights_tohack:
...     w[:] = (w - w.min()) / (w.max() - w.min())
>>> weights_hacked = weights_tohack.transpose(1, 0, 2, 3)[None, :]
>>> conv.weight.data[:] = torch.FloatTensor(weights_hacked)
```

```
>>> plot_convolutional_features(conv, colorspace='rgb', fnum=1, voxels=True, alpha=.
↪6)
```

```
>>> plot_convolutional_features(conv, colorspace='rgb', fnum=2, voxels=False,␣
↪alpha=.9)
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torchvision
>>> model = torchvision.models.resnet50(pretrained=True)
>>> conv = model.conv1
>>> plot_convolutional_features(conv, colorspace='rgb', fnum=None)
```

2.1.3 kwplot.mpl_3d module

Helper for making 3D plots

kwplot.mpl_3d.**plot_surface3d**(*xgrid*, *ygrid*, *zdata*, *xlabel=None*, *ylabel=None*, *zlabel=None*, *wire=False*, *mode=None*, *contour=False*, *rstride=1*, *cstride=1*, *pnum=None*, *labelkw=None*, *xlabelkw=None*, *ylabelkw=None*, *zlabelkw=None*, *titlekw=None*, **args*, ***kwargs*)

References

http://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html

Example

```
>>> # DISABLE_DOCTEST
>>> import kwplot
>>> import matplotlib as mpl
>>> import kwimage
>>> shape=(19, 19)
>>> sigma1, sigma2 = 2.0, 1.0
>>> ybasis = np.arange(shape[0])
>>> xbasis = np.arange(shape[1])
>>> xgrid, ygrid = np.meshgrid(xbasis, ybasis)
>>> sigma = [sigma1, sigma2]
>>> gausspatch = kwimage.gaussian_patch(shape, sigma=sigma)
>>> title = 'ksize={!r}, sigma={!r}'.format(shape, (sigma1, sigma2))
>>> kwplot.plot_surface3d(xgrid, ygrid, gausspatch, rstride=1, cstride=1,
>>>                       cmap=mpl.cm.coolwarm, title=title)
>>> kwplot.show_if_requested()
```

2.1.4 kwplot.mpl_color module

DEPRECATED: use kwimage.Color instead

class kwplot.mpl_color.**Color**(*color*, *alpha=None*, *space=None*)

Bases: NiceRepr

Used for converting a single color between spaces and encodings. This should only be used when handling small numbers of colors(e.g. 1), don't use this to represent an image.

move to colorutil?

Parameters

space (*str*) – colorspace of wrapped color. Assume RGB if not specified and it cannot be inferred

CommandLine

```
xdoctest -m ~/code/kwimage/kwimage/im_color.py Color
```

Example

```
>>> print(Color('g'))
>>> print(Color('orangered'))
>>> print(Color('#AAAAAA').as255())
>>> print(Color([0, 255, 0]))
>>> print(Color([1, 1, 1.]))
>>> print(Color([1, 1, 1]))
>>> print(Color(Color([1, 1, 1])).as255())
```

(continues on next page)

(continued from previous page)

```
>>> print(Color(Color([1., 0, 1, 0])).ashex())
>>> print(Color([1, 1, 1], alpha=255))
>>> print(Color([1, 1, 1], alpha=255, space='lab'))
```

ashex(*space=None*)

as255(*space=None*)

as01(*space=None*)

self = mplutil.Color('red') mplutil.Color('green').as01('rgba')

classmethod named_colors()

Returns
names of colors that Color accepts

Return type
List[str]

Example

```
>>> import kwimage
>>> named_colors = kwimage.Color.named_colors()
>>> color_lut = {name: kwimage.Color(name).as01() for name in named_colors}
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwplot
>>> kwplot.autompl()
>>> canvas = kwplot.make_legend_img(color_lut)
>>> kwplot.imshow(canvas)
```

classmethod distinct(*num*, *existing=None*, *space='rgb'*, *legacy='auto'*, *exclude_black=True*, *exclude_white=True*)

Make multiple distinct colors

References

https://stackoverflow.com/questions/470690/how-to-automatically-generate-n-distinct-colors

Example

```
>>> # xdoctest: +REQUIRES(module:matplotlib)
>>> from kwimage.im_color import *  # NOQA
>>> from kwimage.im_color import _draw_color_swatch
>>> import kwimage
>>> colors1 = kwimage.Color.distinct(10, legacy=False)
>>> swatch1 = _draw_color_swatch(colors1, cellshape=9)
>>> colors2 = kwimage.Color.distinct(10, existing=colors1)
>>> swatch2 = _draw_color_swatch(colors1 + colors2, cellshape=9)
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwplot
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.autompl()
>>> kwplot.imshow(swatch1, pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(swatch2, pnum=(1, 2, 2), fnum=1)
```

classmethod random(*pool='named'*)

distance(*other*, *space='lab'*)

Distance between self an another color

Example

import kwimage self = kwimage.Color((0.16304347826086973, 0.0, 1.0)) other = kwimage.Color('purple')

hard_coded_colors = {
'a': (1.0, 0.0, 0.16), 'b': (1.0, 0.918918918918919, 0.0), 'c': (0.0, 1.0, 0.0), 'd': (0.0, 0.9239130434782604, 1.0), 'e': (0.16304347826086973, 0.0, 1.0)

}

Find grays names = kwimage.Color.named_colors() grays = {} for name in names:

color = kwimage.Color(name) r, g, b = color.as01() if r == g and g == b:

grays[name] = (r, g, b)

print(ub.repr2(ub.sorted_vals(grays), nl=-1))

for k, v in hard_coded_colors.items():
self = kwimage.Color(v) distances = [] for name in names:

other = kwimage.Color(name) dist = self.distance(other) distances.append(dist)

idxs = ub.argsort(distances)[0:5] dists = list(ub.take(distances, idxs)) names = list(ub.take(names, idxs)) print('k = {!r}'.format(k)) print('names = {!r}'.format(names)) print('dists = {!r}'.format(dists))

2.1.5 kwplot.mpl_core module

Extensions of pyplot functionality. Main improvements are

- *kwplot.mpl_core.figure()* can be called with a specific figure number, plot number, and other attributes like if it needs to be cleared or not.
- *kwplot.mpl_core.imshow()* uses simpler defaults for showing image data. Extra normalization is only added if requested.
- *kwplot.mpl_core.close_figures()* This function closes all open figures, which can be helpful in interactive sessions.

kwplot.mpl_core.**next_fnum**(*new_base=None*)

kwplot.mpl_core.**ensure_fnum**(*fnum*)

kwplot.mpl_core.**figure**(*fnum=None*, *pnum=(1, 1, 1)*, *title=None*, *figtitle=None*, *doclf=False*, *docla=False*, *projection=None*, ***kwargs*)

http://matplotlib.org/users/gridspec.html

Parameters

- **fnum** (*int*) – fignum = figure number
- **pnum** (*int, str, or tuple(int, int, int)*) – plotnum = plot tuple
- **title** (*str*) – (default = None)
- **figtitle** (*None*) – (default = None)
- **docla** (*bool*) – (default = False)
- **doclf** (*bool*) – (default = False)

Returns
fig

Return type
mpl.figure.Figure

Example

```
>>> import kwplot
>>> kwplot.autompl()
>>> import matplotlib.pyplot as plt
>>> fnum = 1
>>> fig = figure(fnum, (2, 2, 1))
>>> fig.gca().text(0.5, 0.5, "ax1", va="center", ha="center")
>>> fig = figure(fnum, (2, 2, 2))
>>> fig.gca().text(0.5, 0.5, "ax2", va="center", ha="center")
>>> show_if_requested()
```

Example

```
>>> import kwplot
>>> kwplot.autompl()
>>> import matplotlib.pyplot as plt
>>> fnum = 1
>>> fig = figure(fnum, (2, 2, 1))
>>> fig.gca().text(0.5, 0.5, "ax1", va="center", ha="center")
>>> fig = figure(fnum, (2, 2, 2))
>>> fig.gca().text(0.5, 0.5, "ax2", va="center", ha="center")
>>> fig = figure(fnum, (2, 4, (1, slice(1, None))))
>>> fig.gca().text(0.5, 0.5, "ax3", va="center", ha="center")
>>> show_if_requested()
```

kwplot.mpl_core.**legend**(*loc='best'*, *fontproperties=None*, *size=None*, *fc='w'*, *alpha=1*, *ax=None*, *handles=None*)

Parameters

- **loc** (*str*) – (default = ‘best’) one of ‘best’, ‘upper right’, ‘upper left’, ‘lower left’, ‘lower right’, ‘right’, ‘center left’, ‘center right’, ‘lower center’, or ‘upper center’.
- **fontproperties** (*None*) – (default = None)
- **size** (*None*) – (default = None)

kwplot.mpl_core.**show_if_requested**(*N=1*)

Used at the end of tests. Handles command line arguments for saving figures

Referencse:
http://stackoverflow.com/questions/4325733/save-a-subplot-in-matplotlib

kwplot.mpl_core.**imshow**(*img*, *fnum=None*, *pnum=None*, *xlabel=None*, *title=None*, *figtitle=None*, *ax=None*, *norm=None*, *cmap=None*, *data_colorbar=False*, *colorspace='rgb'*, *interpolation='nearest'*, *alpha=None*, *show_ticks=False*, ***kwargs*)

Parameters

- **img** (*ndarray*) – image data. Height, Width, and Channel dimensions can either be in standard (H, W, C) format or in (C, H, W) format. If C in [3, 4], we assume data is in the rgb / rgba colorspace by default.
- **colorspace** (*str*) – if the data is 3-4 channels, this indicates the colorspace 1 channel data is assumed grayscale. 4 channels assumes alpha.
- **interpolation** (*str*) – either nearest (default), bicubic, bilinear
- **norm** (*bool*) – if True, normalizes the image intensities to fit in a colormap.
- **cmap** (*mpl.colors.Colormap | None*) – color map used if data is not starndard image data
- **data_colorbar** (*bool*) – if True, displays a color scale indicating how colors map to image intensities.
- **fnum** (*int | None*) – figure number
- **pnum** (*tuple | None*) – plot number
- **xlabel** (*str | None*) – sets the label for the x axis
- **title** (*str | None*) – set axes title (if ax is not given)
- **figtitle** (*str | None*) – set figure title (if ax is not given)
- **ax** (*mpl.axes.Axes | None*) – axes to draw on (alternative to fnum and pnum)
- ****kwargs** – docla, doclf, projection

Returns
(fig, ax)

Return type
tuple

kwplot.mpl_core.**set_figtitle**(*figtitle*, *subtitle=''*, *forcefignum=True*, *incanvas=True*, *size=None*, *fontfamily=None*, *fontweight=None*, *fig=None*)

A wrapper around subtitle that also sets the canvas window title if using a Qt backend.

Parameters

- **figtitle** (*str*)
- **subtitle** (*str*)
- **forcefignum** (*bool*) – (default = True)
- **incanvas** (*bool*) – (default = True)
- **fontfamily** (*None*) – (default = None)
- **fontweight** (*None*) – (default = None)
- **size** (*None*) – (default = None)

- **fig** (*None*) – (default = None)

CommandLine

```
python -m kwplot.mpl_core set_figtitle --show
```

Example

```
>>> # DISABLE_DOCTEST
>>> autompl()
>>> fig = figure(fnum=1, doclf=True)
>>> result = set_figtitle(figtitle='figtitle', fig=fig)
>>> # xdoc: +REQUIRES(--show)
>>> show_if_requested()
```

kwplot.mpl_core.**distinct_markers**(*num*, *style='astrisk'*, *total=None*, *offset=0*)

Creates distinct marker codes (as best as possible)

Parameters

- **num** (*int*) – number of markers to make
- **style** (*str*) – mplt style code
- **total** (*int*) – alternative to num
- **offset** (*float*) – angle offset

Returns
marker codes

Return type
List[Tuple]

Example

```
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> style = 'astrisk'
>>> marker_list = kwplot.distinct_markers(10, style)
>>> print('marker_list = {}'.format(ub.repr2(marker_list, nl=1)))
>>> x_data = np.arange(0, 3)
>>> for count, (marker) in enumerate(marker_list):
>>>     plt.plot(x_data, [count] * len(x_data), marker=marker, markersize=10,␣
↪linestyle='', label=str(marker))
>>> plt.legend()
>>> kwplot.show_if_requested()
```

kwplot.mpl_core.**distinct_colors**(*N*, *brightness=0.878*, *randomize=True*, *hue_range=(0.0, 1.0)*, *cmap_seed=None*)

Parameters

- **N** (*int*)

- **brightness** (*float*)

Returns
RGB_tuples

Return type
list

Todo:

- [ ] This is VERY old code that needs massive cleanup.

CommandLine

```
python -m color_funcs --test-distinct_colors --N 2 --show --hue-range=0.05,.95
python -m color_funcs --test-distinct_colors --N 3 --show --hue-range=0.05,.95
python -m color_funcs --test-distinct_colors --N 4 --show --hue-range=0.05,.95
python -m .color_funcs --test-distinct_colors --N 3 --show --no-randomize
python -m .color_funcs --test-distinct_colors --N 4 --show --no-randomize
python -m .color_funcs --test-distinct_colors --N 6 --show --no-randomize
python -m .color_funcs --test-distinct_colors --N 20 --show
```

References

http://blog.jianhuashao.com/2011/09/generate-n-distinct-colors.html

CommandLine

```
python -m .color_funcs --exec-distinct_colors --show
python -m .color_funcs --exec-distinct_colors --show --no-randomize --N 50
python -m .color_funcs --exec-distinct_colors --show --cmap_seed=foobar
```

kwplot.mpl_core.**phantom_legend**(*label_to_color*, *mode='line'*, *ax=None*, *legend_id=None*, *loc=0*)

Creates a legend on an axis based on a label-to-color map.

Parameters
label_to_color (*Dict[str, kwimage.Color]*) – mapping from string label to the color.

Todo:

- [ ] More docs and ensure this exists in the right place

kwplot.mpl_core.**close_figures**(*figures=None*)

Close specified figures. If no figures are specified, close all figure.

Parameters
figures (*List[mpl.figure.Figure]*) – list of figures to close

kwplot.mpl_core.**all_figures**()

Return a list of all open figures

Returns
list of all figures

Return type
List[mpl.figure.Figure]

2.1.6 kwplot.mpl_draw module

Note, this module should be refactored into MPL figure drawings and cv2 on-image drawings.

kwplot.mpl_draw.**draw_boxes**(*boxes*, *alpha=None*, *color='blue'*, *labels=None*, *centers=False*, *fill=False*, *ax=None*, *lw=2*)

Parameters

- **boxes** (*kwimage.Boxes*)
- **labels** (*List[str]*) – of labels
- **alpha** (*List[float]*) – alpha for each box
- **centers** (*bool*) – draw centers or not
- **lw** (*float*) – linewidth

Example

```
>>> import kwimage
>>> bboxes = kwimage.Boxes([[.1, .1, .6, .3], [.3, .5, .5, .6]], 'xywh')
>>> draw_boxes(bboxes)
>>> #kwplot.autompl()
```

kwplot.mpl_draw.**draw_line_segments**(*pts1*, *pts2*, *ax=None*, ***kwargs*)

draws *N* line segments between *N* pairs of points

Parameters

- **pts1** (*ndarray*) – Nx2
- **pts2** (*ndarray*) – Nx2
- **ax** (*None*) – (default = None)
- ****kwargs** – lw, alpha, colors

Example

```
>>> import numpy as np
>>> import kwplot
>>> pts1 = np.array([(.1, .8), (.6, .8)])
>>> pts2 = np.array([(.6, .7), (.4, .1)])
>>> kwplot.figure(fnum=None)
>>> draw_line_segments(pts1, pts2)
>>> # xdoc: +REQUIRES(--show)
>>> import matplotlib.pyplot as plt
>>> ax = plt.gca()
```

(continues on next page)

(continued from previous page)

```
>>> ax.set_xlim(0, 1)
>>> ax.set_ylim(0, 1)
>>> kwplot.show_if_requested()
```

kwplot.mpl_draw.**plot_matrix**(*matrix*, *index=None*, *columns=None*, *rot=90*, *ax=None*, *grid=True*, *label=None*, *zerodiag=False*, *cmap='viridis'*, *showvals=False*, *showzero=True*, *logscale=False*, *xlabel=None*, *ylabel=None*, *fnum=None*, *pnum=None*)

Helper for plotting confusion matrices

Parameters

matrix (*ndarray | pd.DataFrame*) – if a data frame then index, columns, xlabel, and ylabel will be defaulted to sensible values.

Todo:

- [ ] Finish args docs
- [ ] Replace internals with seaborn

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwplot.mpl_draw import *  # NOQA
>>> import pandas as pd
>>> classes = ['cls1', 'cls2', 'cls3']
>>> matrix = np.array([[2, 2, 1], [3, 1, 0], [1, 0, 0]])
>>> matrix = pd.DataFrame(matrix, index=classes, columns=classes)
>>> matrix.index.name = 'real'
>>> matrix.columns.name = 'pred'
>>> plot_matrix(matrix, showvals=True)
>>> # xdoc: +REQUIRES(--show)
>>> import matplotlib.pyplot as plt
>>> import kwplot
>>> kwplot.autompl()
>>> plot_matrix(matrix, showvals=True)
```

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwplot.mpl_draw import *  # NOQA
>>> matrix = np.array([[2, 2, 1], [3, 1, 0], [1, 0, 0]])
>>> plot_matrix(matrix)
>>> # xdoc: +REQUIRES(--show)
>>> import matplotlib.pyplot as plt
>>> import kwplot
>>> kwplot.autompl()
>>> plot_matrix(matrix)
```

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwplot.mpl_draw import *  # NOQA
>>> matrix = np.array([[2, 2, 1], [3, 1, 0], [1, 0, 0]])
>>> classes = ['cls1', 'cls2', 'cls3']
>>> plot_matrix(matrix, index=classes, columns=classes)
```

kwplot.mpl_draw.**draw_points**(*xy*, *color='blue'*, *class_idxs=None*, *classes=None*, *ax=None*, *alpha=None*, *radius=1*, ***kwargs*)

Parameters

xy (*ndarray*) – of points.

Example

```
>>> from kwplot.mpl_draw import *  # NOQA
>>> import kwimage
>>> xy = kwimage.Points.random(10).xy
>>> draw_points(xy, radius=0.01)
>>> draw_points(xy, class_idxs=np.random.randint(0, 3, 10),
>>>         radius=0.01, classes=['a', 'b', 'c'], color='classes')
```

kwplot.mpl_draw.**draw_text_on_image**(*img*, *text*, *org=None*, *return_info=False*, ***kwargs*)

Draws multiline text on an image using opencv

Parameters

- **img** (*ndarray | None | dict*) – Generally a numpy image to draw on (inplace). Otherwise a canvas will be constructed such that the text will fit. The user may specify a dictionary with keys width and height to have more control over the constructed canvas.
- **text** (*str*) – text to draw
- **org** (*Tuple[int, int]*) – The x, y location of the text string "anchor" in the image as specified by halign and valign. For instance, If valign='bottom', halign='left', this is the bottom left corner.
- **return_info** (*bool, default=False*) – if True, also returns information about the positions the text was drawn on.
- ****kwargs** – color (tuple): default blue thickness (int): defaults to 2 fontFace (int): defaults to cv2.FONT_HERSHEY_SIMPLEX fontScale (float): defaults to 1.0 valign (str, default='bottom'):

  either top, center, or bottom. NOTE: this default may change to "top" in the future.

  halign (str, default='left'):
  either left, center, or right

  border (dict | int):
  If specified as an integer, draws a black border with that given thickness. If specified as a dictionary, draws a border with color specified parameters.

  "color": border color, defaults to "black". "thickness": border thickness, defaults to 1.

Returns
the image that was drawn on

Return type
ndarray

Note: The image is modified inplace. If the image is non-contiguous then this returns a UMat instead of a ndarray, so be carefull with that.

References

https://stackoverflow.com/questions/27647424/ https://stackoverflow.com/questions/51285616/opencvs-gettextsize-and-puttext-return-wrong-size-and-chop-letters-with-low

Example

```
>>> import kwimage
>>> img = kwimage.grab_test_image(space='rgb')
>>> img2 = kwimage.draw_text_on_image(img.copy(), 'FOOBAR', org=(0, 0), valign='top
↪')
>>> assert img2.shape == img.shape
>>> assert np.any(img2 != img)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img2)
>>> kwplot.show_if_requested()
```

Example

```
>>> import kwimage
>>> # Test valign
>>> img = kwimage.grab_test_image(space='rgb', dsize=(500, 500))
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(0, 0),␣
↪valign='top', border=2)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(150, 0),␣
↪valign='center', border=2)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(300, 0),␣
↪valign='bottom', border=2)
>>> # Test halign
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(250, 100),␣
↪halign='right', border=2)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(250, 250),␣
↪halign='center', border=2)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(250, 400),␣
↪halign='left', border=2)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.imshow(img2)
>>> kwplot.show_if_requested()
```

Example

```
>>> # Ensure the function works with float01 or uint255 images
>>> import kwimage
>>> img = kwimage.grab_test_image(space='rgb')
>>> img = kwimage.ensure_float01(img)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(0, 0),␣
↪valign='top', border=2)
```

Example

```
>>> # Test dictionary border
>>> import kwimage
>>> img = kwimage.draw_text_on_image(None, 'hello\neveryone', org=(100, 100),␣
↪valign='top', halign='center', border={'color': 'green', 'thickness': 9})
>>> #img = kwimage.draw_text_on_image(None, 'hello\neveryone', org=(0, 0), valign='top
↪')
>>> #img = kwimage.draw_text_on_image(None, 'hello', org=(0, 60), valign='top', halign=
↪'center', border=0)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img)
>>> kwplot.show_if_requested()
```

Example

```
>>> # Test dictionary image
>>> import kwimage
>>> img = kwimage.draw_text_on_image({'width': 300}, 'good\nPropogate', org=(150,␣
↪0), valign='top', halign='center', border={'color': 'green', 'thickness': 0})
>>> print('img.shape = {!r}'.format(img.shape))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img)
>>> kwplot.show_if_requested()
```

Example

```
>>> import ubelt as ub
>>> import kwimage
>>> grid = list(ub.named_product({
>>>     'halign': ['left', 'center', 'right', None],
>>>     'valign': ['top', 'center', 'bottom', None],
>>>     'border': [0, 3]
>>> }))
>>> canvases = []
>>> text = 'small-line\na-much-much-much-bigger-line\nanother-small\n.'
>>> for kw in grid:
>>>     header = kwimage.draw_text_on_image({}, ub.repr2(kw, compact=1), color='blue
↪')
>>>     canvas = kwimage.draw_text_on_image({'color': 'white'}, text, org=None,␣
↪**kw)
>>>     canvases.append(kwimage.stack_images([header, canvas], axis=0, bg_
↪value=(255, 255, 255), pad=5))
>>> # xdoc: +REQUIRES(--show)
>>> canvas = kwimage.stack_images_grid(canvases, pad=10, bg_value=(255, 255, 255))
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(canvas)
>>> kwplot.show_if_requested()
```

kwplot.mpl_draw.**draw_boxes_on_image**(*img*, *boxes*, *color='blue'*, *thickness=1*, *box_format=None*, *colorspace='rgb'*)

Draws boxes on an image.

Parameters

- **img** (*ndarray*) – image to copy and draw on
- **boxes** (*nh.util.Boxes*) – boxes to draw
- **colorspace** (*str*) – string code of the input image colorspace

Example

```
>>> import kwimage
>>> import numpy as np
>>> img = np.zeros((10, 10, 3), dtype=np.uint8)
>>> color = 'dodgerblue'
>>> thickness = 1
>>> boxes = kwimage.Boxes([[1, 1, 8, 8]], 'ltrb')
>>> img2 = draw_boxes_on_image(img, boxes, color, thickness)
>>> assert tuple(img2[1, 1]) == (30, 144, 255)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()  # xdoc: +SKIP
>>> kwplot.figure(doclf=True, fnum=1)
>>> kwplot.imshow(img2)
```

kwplot.mpl_draw.**draw_clf_on_image**(*im*, *classes*, *tcx=None*, *probs=None*, *pcx=None*, *border=1*)

Draws classification label on an image.

Works best with image chips sized between 200x200 and 500x500

Parameters

- **im** (*ndarray*) – the image
- **classes** (*Sequence | CategoryTree*) – list of class names
- **tcx** (*int, default=None*) – true class index if known
- **probs** (*ndarray*) – predicted class probs for each class
- **pcx** (*int, default=None*) – predicted class index. (if None but probs is specified uses argmax of probs)

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torch
>>> import kwarray
>>> import kwimage
>>> rng = kwarray.ensure_rng(0)
>>> im = (rng.rand(300, 300) * 255).astype(np.uint8)
>>> classes = ['cls_a', 'cls_b', 'cls_c']
>>> tcx = 1
>>> probs = rng.rand(len(classes))
>>> probs[tcx] = 0
>>> probs = torch.FloatTensor(probs).softmax(dim=0).numpy()
>>> im1_ = kwimage.draw_clf_on_image(im, classes, tcx, probs)
>>> probs[tcx] = .9
>>> probs = torch.FloatTensor(probs).softmax(dim=0).numpy()
>>> im2_ = kwimage.draw_clf_on_image(im, classes, tcx, probs)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(im1_, colorspace='rgb', pnum=(1, 2, 1), fnum=1, doclf=True)
>>> kwplot.imshow(im2_, colorspace='rgb', pnum=(1, 2, 2), fnum=1)
>>> kwplot.show_if_requested()
```

2.1.7 kwplot.mpl_make module

DEPRECATED: use kwimage versions instead

Functions used to explicitly make images as ndarrays using mpl/cv2 utilities

kwplot.mpl_make.**make_heatmask**(*probs*, *cmap='plasma'*, *with_alpha=1.0*, *space='rgb'*, *dsize=None*)

Colorizes a single-channel intensity mask (with an alpha channel)

Parameters

- **probs** (*ndarray*) – 2D probability map with values between 0 and 1
- **cmap** (*str*) – mpl colormap
- **with_alpha** (*float*) – between 0 and 1, uses probs as the alpha multipled by this number.

- **space** (*str*) – output colorspace
- **dsize** (*tuple*) – if not None, then output is resized to W,H=dsize

SeeAlso:
kwimage.overlay_alpha_images

Example

```
>>> # xdoc: +REQUIRES(module:matplotlib)
>>> from kwimage.im_draw import *  # NOQA
>>> probs = np.tile(np.linspace(0, 1, 10), (10, 1))
>>> heatmask = make_heatmask(probs, with_alpha=0.8, dsize=(100, 100))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.imshow(heatmask, fnum=1, doclf=True, colorspace='rgb')
>>> kwplot.show_if_requested()
```

kwplot.mpl_make.**make_vector_field**(*dx*, *dy*, *stride=0.02*, *thresh=0.0*, *scale=1.0*, *alpha=1.0*, *color='strawberry'*, *thickness=1*, *tipLength=0.1*, *line_type='aa'*)

Create an image representing a 2D vector field.

Parameters

- **dx** (*ndarray*) – grid of vector x components
- **dy** (*ndarray*) – grid of vector y components
- **stride** (*int | float*) – sparsity of vectors, int specifies stride step in pixels, a float specifies it as a percentage.
- **thresh** (*float*) – only plot vectors with magnitude greater than thres
- **scale** (*float*) – multiply magnitude for easier visualization
- **alpha** (*float*) – alpha value for vectors. Non-vector regions receive 0 alpha (if False, no alpha channel is used)
- **color** (*str | tuple | kwimage.Color*) – RGB color of the vectors
- **thickness** (*int, default=1*) – thickness of arrows
- **tipLength** (*float, default=0.1*) – fraction of line length
- **line_type** (*int*) – either cv2.LINE_4, cv2.LINE_8, or cv2.LINE_AA

Returns
vec_img: an rgb/rgba image in 0-1 space

Return type
ndarray[float32]

SeeAlso:
kwimage.overlay_alpha_images

DEPRECATED USE: draw_vector_field instead

Example

```
>>> x, y = np.meshgrid(np.arange(512), np.arange(512))
>>> dx, dy = x - 256.01, y - 256.01
>>> radians = np.arctan2(dx, dy)
>>> mag = np.sqrt(dx ** 2 + dy ** 2)
>>> dx, dy = dx / mag, dy / mag
>>> img = make_vector_field(dx, dy, scale=10, alpha=False)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img)
>>> kwplot.show_if_requested()
```

kwplot.mpl_make.**make_orimask**(*radians*, *mag=None*, *alpha=1.0*)

Makes a colormap in HSV space where the orientation changes color and mag changes the saturation/value.

Parameters

- **radians** (*ndarray*) – orientation in radians
- **mag** (*ndarray*) – magnitude (must be normalized between 0 and 1)
- **alpha** (*float | ndarray*) – if False or None, then the image is returned without alpha if a float, then mag is scaled by this and used as the alpha channel if an ndarray, then this is explicilty set as the alpha channel

Returns

an rgb / rgba image in 01 space

Return type

ndarray[float32]

SeeAlso:

kwimage.overlay_alpha_images

Example

```
>>> # xdoc: +REQUIRES(module:matplotlib)
>>> from kwimage.im_draw import *  # NOQA
>>> x, y = np.meshgrid(np.arange(64), np.arange(64))
>>> dx, dy = x - 32, y - 32
>>> radians = np.arctan2(dx, dy)
>>> mag = np.sqrt(dx ** 2 + dy ** 2)
>>> orimask = make_orimask(radians, mag)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.imshow(orimask, fnum=1, doclf=True, colorspace='rgb')
>>> kwplot.show_if_requested()
```

kwplot.mpl_make.**make_legend_img**(*label_to_color*, *dpi=96*, *shape=(200, 200)*, *mode='line'*, *transparent=False*)

Makes an image of a categorical legend

Parameters

label_to_color (*Dict[str, kwimage.Color]*) – mapping from string label to the color.

CommandLine

```
xdoctest -m kwplot.mpl_make make_legend_img --show
```

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwplot
>>> import kwimage
>>> label_to_color = {
>>>     'blue': kwimage.Color('blue').as01(),
>>>     'red': kwimage.Color('red').as01(),
>>>     'green': 'green',
>>>     'yellow': 'yellow',
>>>     'orangered': 'orangered',
>>> }
>>> img = make_legend_img(label_to_color, transparent=True)
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.autompl()
>>> kwplot.imshow(img)
>>> kwplot.show_if_requested()
```

kwplot.mpl_make.**render_figure_to_image**(*fig*, *dpi=None*, *transparent=None*, ***savekw*)

Saves a figure as an image in memory.

Parameters

- **fig** (*matplotlib.figure.Figure*) – figure to save
- **dpi** (*Optional[int | str]*) – The resolution in dots per inch. If *None* it will default to the value `savefig.dpi` in the matplotlibrc file. If 'figure' it will set the dpi to be the value of the figure.
- **transparent** (*bool*) – If *True*, the axes patches will all be transparent; the figure patch will also be transparent unless facecolor and/or edgecolor are specified via kwargs.
- ****savekw** – other keywords passed to `fig.savefig`. Valid keywords include: facecolor, edgecolor, orientation, papertype, format, pad_inches, frameon.

Returns

an image in RGB or RGBA format.

Return type

np.ndarray

Note: Be sure to use *fig.set_size_inches* to an appropriate size before calling this function.

Example

```
>>> import kwplot
>>> fig = kwplot.figure(fnum=1, doclf=True)
>>> ax = fig.gca()
>>> ax.cla()
>>> ax.plot([0, 10], [0, 10])
>>> canvas_rgb = kwplot.render_figure_to_image(fig, transparent=False)
>>> canvas_rgba = kwplot.render_figure_to_image(fig, transparent=True)
>>> assert canvas_rgb.shape[2] == 3
>>> assert canvas_rgba.shape[2] == 4
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.autompl()
>>> kwplot.imshow(canvas_rgb, fnum=2)
>>> kwplot.show_if_requested()
```

2.1.8 kwplot.mpl_multiplot module

DEPRECATED: Use seaborn instead

kwplot.mpl_multiplot.**multi_plot**(*xdata=None*, *ydata=None*, *xydata=None*, ***kwargs*)

plots multiple lines, bars, etc...

One function call that concisely describes the all of the most commonly used parameters needed when plotting a bar / line char. This is especially useful when multiple plots are needed in the same domain.

Parameters

- **xdata** (*List[ndarray] | Dict[str, ndarray] | ndarray*) – x-coordinate data common to all y-coordinate values or xdata for each line/bar in ydata. Mutually exclusive with xydata.
- **ydata** (*List[ndarray] | Dict[str, ndarray] | ndarray*) – y-coordinate values for each line/bar to plot. Can also be just a single ndarray of scalar values. Mutually exclusive with xydata.
- **xydata** (*Dict[str, Tuple[ndarray, ndarray]]*) – mapping from labels to a tuple of xdata and ydata for a each line.
- ****kwargs** –

  fnum (int):
  figure number to draw on

  pnum (Tuple[int, int, int]):
  plot number to draw on within the figure, e.g. (1, 1, 1)

  label (List|Dict):
  if you specified ydata as a List[ndarray] this is the label for each line in that list. Note this is unnecessary if you specify input as a dictionary mapping labels to lines.

  color (str|List|Dict):
  either a special color code, a single color, or a color for each item in ydata. In the later case, this should be specified as either a list or a dict depending on how ydata was specified.

  marker (str|List|Dict):
  type of matplotlib marker to use at every data point. Can be specified for all lines jointly or for each line independently.

  transpose (bool, default=False):
  swaps x and y data.

kind (str, default='plot'):
The kind of plot. Can either be 'plot' or 'bar'. We parse these other kwargs if:

if kind='plot':
spread

if kind='bar':
stacked, width

Misc:
use_legend (bool): … legend_loc (str):

one of 'best', 'upper right', 'upper left', 'lower left', 'lower right', 'right', 'center left', 'center right', 'lower center', or 'upper center'.

Layout:
xlabel (str): label for x-axis ylabel (str): label for y-axis title (str): title for the axes figtitle (str): title for the figure

xscale (str): can be one of [linear, log, logit, symlog] yscale (str): can be one of [linear, log, logit, symlog]

xlim (Tuple[float, float]): low and high x-limit of axes ylim (Tuple[float, float]): low and high y-limit of axes xmin (float): low x-limit of axes, mutex with xlim xmax (float): high x-limit of axes, mutex with xlim ymin (float): low y-limit of axes, mutex with ylim ymax (float): high y-limit of axes, mutex with ylim

titlesize (float): … legendsize (float): … labelsize (float): …

Grid:
gridlinewidth (float): … gridlinestyle (str): …

Ticks:
num_xticks (int): number of x ticks num_yticks (int): number of y ticks tickwidth (float): … ticklength (float): … ticksize (float): … xticklabels (list): list of x-tick labels, overrides num_xticks yticklabels (list): list of y-tick labels, overrides num_yticks xtick_rotation (float): xtick rotation in degrees ytick_rotation (float): ytick rotation in degrees

Data:

spread (List | Dict): Plots a spread around plot lines usually
indicating standard deviation

markersize (float|List|Dict): marker size for all or each plot markeredgewidth (float|List|Dict): marker edge width for all or each plot linewidth (float|List|Dict): line width for all or each plot linestyle (str|List|Dict): line style for all or each plot

---

Note: any plot_kw key can be a scalar (corresponding to all ydatas), a list if ydata was specified as a list, or a dict if ydata was specified as a dict.

plot_kw_keys = ['label', 'color', 'marker', 'markersize',
'markeredgewidth', 'linewidth', 'linestyle']

Note: In general this should be deprecated in favor of using seaborn

---

Returns
ax : the axes that was drawn on

Return type
matplotlib.axes.Axes

References

matplotlib.org/examples/api/barchart_demo.html

Example

```
>>> import kwplot
>>> kwplot.autompl()
>>> # The new way to use multi_plot is to pass ydata as a dict of lists
>>> ydata = {
>>>     'spam': [1, 1, 2, 3, 5, 8, 13],
>>>     'eggs': [3, 3, 3, 3, 3, np.nan, np.nan],
>>>     'jamμ': [5, 3, np.nan, 1, 2, np.nan, np.nan],
>>>     'pram': [4, 2, np.nan, 0, 0, np.nan, 1],
>>> }
>>> ax = kwplot.multi_plot(ydata=ydata, title='μμμ',
>>>                        xlabel='\nfdsμμμ', linestyle='--')
>>> kwplot.show_if_requested()
```

Example

```
>>> # Old way to use multi_plot is a list of lists
>>> import kwplot
>>> kwplot.autompl()
>>> xdata = [1, 2, 3, 4, 5]
>>> ydata_list = [[1, 2, 3, 4, 5], [3, 3, 3, 3, 3], [5, 4, np.nan, 2, 1], [4, 3, np.
↪nan, 1, 0]]
>>> kwargs = {'label': ['spam', 'eggs', 'jamμ', 'pram'],  'linestyle': '-'}
>>> #ax = multi_plot(xdata, ydata_list, title='$\phi_1(\\vec{x})$', xlabel='\nfds',␣
↪**kwargs)
>>> ax = multi_plot(xdata, ydata_list, title='μμμ', xlabel='\nfdsμμμ', **kwargs)
>>> kwplot.show_if_requested()
```

Example

```
>>> # Simple way to use multi_plot is to pass xdata and ydata exactly
>>> # like you would use plt.plot
>>> import kwplot
>>> kwplot.autompl()
>>> ax = multi_plot([1, 2, 3], [4, 5, 6], fnum=4, label='foo')
>>> kwplot.show_if_requested()
```

Example

```
>>> import kwplot
>>> kwplot.autompl()
>>> xydata = {'a': ([0, 1, 2], [0, 1, 2]), 'b': ([0, 2, 4], [2, 1, 0])}
>>> ax = kwplot.multi_plot(xydata=xydata, fnum=4)
>>> kwplot.show_if_requested()
```

Example

```
>>> import kwplot
>>> kwplot.autompl()
>>> ydata = {'a': [0, 1, 2], 'b': [1, 2, 1], 'c': [4, 4, 4, 3, 2]}
>>> kwargs = {
>>>     'spread': {'a': [.2, .3, .1], 'b': .2},
>>>     'xlim': (-1, 5),
>>>     'xticklabels': ['foo', 'bar'],
>>>     'xtick_rotation': 90,
>>> }
>>> ax = kwplot.multi_plot(ydata=ydata, fnum=4, **kwargs)
>>> kwplot.show_if_requested()
```

2.1.9 kwplot.mpl_plotnums module

Defines the *kwplot.mpl_plotnums.PlotNums* class to help manage a grid of subplot numbers.

class kwplot.mpl_plotnums.**PlotNums**(*nRows=None*, *nCols=None*, *nSubplots=None*, *start=0*)

Bases: object

Convinience class for dealing with plot numberings (pnums)

This is useful in the case where you want a certain number of subplots, but you might swap the order in which those subplots are called. This class introduces the idea of either getting the "next" subplot, or getting one at a specific instance. The total number of subplots can be modified in just a single place in the code (the arguments to the PlotNums constructor) instead of each instance where you would specify a pnum normally.

Example

```
>>> import ubelt as ub
>>> pnum_ = PlotNums(nRows=2, nCols=2)
>>> # Indexable
>>> print(pnum_[0])
(2, 2, 1)
>>> # Iterable
>>> print(ub.repr2(list(pnum_), nl=0, nobr=1))
(2, 2, 1), (2, 2, 2), (2, 2, 3), (2, 2, 4)
>>> # Callable (iterates through a default iterator)
>>> print(pnum_())
(2, 2, 1)
>>> print(pnum_())
(2, 2, 2)
```

2.2 Module contents

2.2.1 KWPlot - The Kitware Plot Module

This module is a small wrapper around matplotlib and seaborn that simplifies develoepr workflow when working with code that might be run in IPython or in a script. This is primarilly handled by the *kwplot.auto_backends* module, which exposes the functions: *kwplot.autompl()*, *kwplot.autoplt()*, and *kwplot.autosns()* for auto-initialization of matplotlib, pyplot, and seaborn.

A very common anti-pattern in developer code is importing `matplotlib.pyplot` at the top level of your module. This is a mistake because importing pyplot has side-effects which can cause problems if executed at a module level (i.e. they happen at import time! Anyone using your library will have to deal with these consequences)

To mitigate this we recommend only using pyplot inside of the scope of the functions that need it.

Importing *kwplot* itself has no import-time side effects, so it is safe to put it as a module level import, however, plotting is often an optional feature of any library, so we still recomend putting that code inside the functions that need it.

The general code flow looks like this, inside your function run:

```
import kwplot
kwplot.autompl()

# Pyplot is now initialized do matplotlib or pyplot stuff
...
```

This checks if you are running interactively in IPython, if so try to use a Qt backend. If not, then try to use a headless Agg backend.

You can also do

```
import kwplot
# These also call autompl in the backend, and return either the seaborn or
# pyplot modules, so you dont have to import them in your code. When
# running seaborn, this will also call ``sns.set()`` for you.
sns = kwplot.autosns()
plt = kwplot.autoplt()
...
```

In addition to this auto-backend feature, kwplot also exposes useful helper methods for common drawing operations.

There is also a small CLI that can be used to view multispectral or uint16 images.

class kwplot.**BackendContext**(*backend*)

Bases: `object`

Context manager that ensures a specific backend, but then reverts after the context has ended.

Because this changes the backend after pyplot has initialized, there is a chance for odd behavior to occur. Please submit and issue if you experience this and can document the environment that caused it.

CommandLine

```
# Checks
xdoctest -m kwplot.auto_backends BackendContext --check
```

Example

```
>>> # xdoctest +REQUIRES(--check)
>>> from kwplot.auto_backends import *  # NOQA
>>> import matplotlib as mpl
>>> import kwplot
>>> print(mpl.get_backend())
>>> #kwplot.autompl(force='auto')
>>> #print(mpl.get_backend())
>>> #fig1 = kwplot.figure(fnum=3)
>>> #print(mpl.get_backend())
>>> with BackendContext('agg'):
>>>     print(mpl.get_backend())
>>>     fig2 = kwplot.figure(fnum=4)
>>> print(mpl.get_backend())
```

class kwplot.**Color**(*color*, *alpha=None*, *space=None*)

Bases: NiceRepr

Used for converting a single color between spaces and encodings. This should only be used when handling small numbers of colors(e.g. 1), don't use this to represent an image.

move to colorutil?

Parameters

space (*str*) – colorspace of wrapped color. Assume RGB if not specified and it cannot be inferred

CommandLine

```
xdoctest -m ~/code/kwimage/kwimage/im_color.py Color
```

Example

```
>>> print(Color('g'))
>>> print(Color('orangered'))
>>> print(Color('#AAAAAA').as255())
>>> print(Color([0, 255, 0]))
>>> print(Color([1, 1, 1.]))
>>> print(Color([1, 1, 1]))
>>> print(Color(Color([1, 1, 1])).as255())
>>> print(Color(Color([1., 0, 1, 0])).ashex())
>>> print(Color([1, 1, 1], alpha=255))
>>> print(Color([1, 1, 1], alpha=255, space='lab'))
```

ashex(*space=None*)

as255(*space=None*)

as01(*space=None*)

self = mplutil.Color('red') mplutil.Color('green').as01('rgba')

classmethod named_colors()

Returns
names of colors that Color accepts

Return type
List[str]

Example

```
>>> import kwimage
>>> named_colors = kwimage.Color.named_colors()
>>> color_lut = {name: kwimage.Color(name).as01() for name in named_colors}
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwplot
>>> kwplot.autompl()
>>> canvas = kwplot.make_legend_img(color_lut)
>>> kwplot.imshow(canvas)
```

classmethod distinct(*num*, *existing=None*, *space='rgb'*, *legacy='auto'*, *exclude_black=True*, *exclude_white=True*)

Make multiple distinct colors

References

https://stackoverflow.com/questions/470690/how-to-automatically-generate-n-distinct-colors

Example

```
>>> # xdoctest: +REQUIRES(module:matplotlib)
>>> from kwimage.im_color import *  # NOQA
>>> from kwimage.im_color import _draw_color_swatch
>>> import kwimage
>>> colors1 = kwimage.Color.distinct(10, legacy=False)
>>> swatch1 = _draw_color_swatch(colors1, cellshape=9)
>>> colors2 = kwimage.Color.distinct(10, existing=colors1)
>>> swatch2 = _draw_color_swatch(colors1 + colors2, cellshape=9)
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(swatch1, pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(swatch2, pnum=(1, 2, 2), fnum=1)
```

classmethod random(*pool='named'*)

distance(*other*, *space='lab'*)

Distance between self an another color

Example

import kwimage self = kwimage.Color((0.16304347826086973, 0.0, 1.0)) other = kwimage.Color('purple')

hard_coded_colors = {
'a': (1.0, 0.0, 0.16), 'b': (1.0, 0.918918918918919, 0.0), 'c': (0.0, 1.0, 0.0), 'd': (0.0, 0.9239130434782604, 1.0), 'e': (0.16304347826086973, 0.0, 1.0)

}

Find grays names = kwimage.Color.named_colors() grays = {} for name in names:

color = kwimage.Color(name) r, g, b = color.as01() if r == g and g == b:

grays[name] = (r, g, b)

print(ub.repr2(ub.sorted_vals(grays), nl=-1))

for k, v in hard_coded_colors.items():
self = kwimage.Color(v) distances = [] for name in names:

other = kwimage.Color(name) dist = self.distance(other) distances.append(dist)

idxs = ub.argsort(distances)[0:5] dists = list(ub.take(distances, idxs)) names = list(ub.take(names, idxs)) print('k = {!r}'.format(k)) print('names = {!r}'.format(names)) print('dists = {!r}'.format(dists))

class kwplot.**PlotNums**(*nRows=None*, *nCols=None*, *nSubplots=None*, *start=0*)

Bases: object

Convinience class for dealing with plot numberings (pnums)

This is useful in the case where you want a certain number of subplots, but you might swap the order in which those subplots are called. This class introduces the idea of either getting the "next" subplot, or getting one at a specific instance. The total number of subplots can be modified in just a single place in the code (the arguments to the PlotNums constructor) instead of each instance where you would specify a pnum normally.

Example

```
>>> import ubelt as ub
>>> pnum_ = PlotNums(nRows=2, nCols=2)
>>> # Indexable
>>> print(pnum_[0])
(2, 2, 1)
>>> # Iterable
>>> print(ub.repr2(list(pnum_), nl=0, nobr=1))
(2, 2, 1), (2, 2, 2), (2, 2, 3), (2, 2, 4)
>>> # Callable (iterates through a default iterator)
>>> print(pnum_())
(2, 2, 1)
>>> print(pnum_())
(2, 2, 2)
```

kwplot.**all_figures**()

Return a list of all open figures

Returns
list of all figures

Return type
List[mpl.figure.Figure]

kwplot.**autompl**(*verbose=0*, *recheck=False*, *force=None*)

Uses platform heuristics to automatically set the matplotlib backend. If no display is available it will be set to *agg*, otherwise we will try to use the cross-platform *Qt5Agg* backend.

Parameters

- **verbose** (*int, default=0*) – verbosity level
- **recheck** (*bool, default=False*) – if False, this function will not run if it has already been called (this can save a significant amount of time).
- **force** (*str, default=None*) – backend to force to or "auto"

CommandLine

```
# Checks
export QT_DEBUG_PLUGINS=1
xdoctest -m kwplot.auto_backends autompl --check
KWPLOT_UNSAFE=1 xdoctest -m kwplot.auto_backends autompl --check
KWPLOT_UNSAFE=0 xdoctest -m kwplot.auto_backends autompl --check
```

Example

```
>>> # xdoctest +REQUIRES(--check)
>>> plt = autoplt(verbose=1)
>>> plt.figure()
```

References

https://stackoverflow.com/questions/637005/check-if-x-server-is-running

kwplot.**autoplt**(*verbose=0*, *recheck=False*, *force=None*)

Like autompl, but also returns the `matplotlib.pyplot` module for convenience.

Returns
ModuleType

kwplot.**autosns**(*verbose=0*, *recheck=False*, *force=None*)

Like autompl, but also calls `seaborn.set()` and returns the `seaborn` module for convenience.

Returns
ModuleType

kwplot.**close_figures**(*figures=None*)

Close specified figures. If no figures are specified, close all figure.

Parameters
figures (*List[mpl.figure.Figure]*) – list of figures to close

kwplot.**distinct_colors**(*N*, *brightness=0.878*, *randomize=True*, *hue_range=(0.0, 1.0)*, *cmap_seed=None*)

Parameters

- **N** (*int*)
- **brightness** (*float*)

Returns
RGB_tuples

Return type
list

Todo:

- [ ] This is VERY old code that needs massive cleanup.

CommandLine

```
python -m color_funcs --test-distinct_colors --N 2 --show --hue-range=0.05,.95
python -m color_funcs --test-distinct_colors --N 3 --show --hue-range=0.05,.95
python -m color_funcs --test-distinct_colors --N 4 --show --hue-range=0.05,.95
python -m .color_funcs --test-distinct_colors --N 3 --show --no-randomize
python -m .color_funcs --test-distinct_colors --N 4 --show --no-randomize
python -m .color_funcs --test-distinct_colors --N 6 --show --no-randomize
python -m .color_funcs --test-distinct_colors --N 20 --show
```

References

http://blog.jianhuashao.com/2011/09/generate-n-distinct-colors.html

CommandLine

```
python -m .color_funcs --exec-distinct_colors --show
python -m .color_funcs --exec-distinct_colors --show --no-randomize --N 50
python -m .color_funcs --exec-distinct_colors --show --cmap_seed=foobar
```

kwplot.**distinct_markers**(*num*, *style='astrisk'*, *total=None*, *offset=0*)

Creates distinct marker codes (as best as possible)

Parameters

- **num** (*int*) – number of markers to make
- **style** (*str*) – mplt style code
- **total** (*int*) – alternative to num

- **offset** (*float*) – angle offset

Returns
marker codes

Return type
List[Tuple]

Example

```
>>> import kwplot
>>> plt = kwplot.autoplt()
>>> style = 'astrisk'
>>> marker_list = kwplot.distinct_markers(10, style)
>>> print('marker_list = {}'.format(ub.repr2(marker_list, nl=1)))
>>> x_data = np.arange(0, 3)
>>> for count, (marker) in enumerate(marker_list):
>>>     plt.plot(x_data, [count] * len(x_data), marker=marker, markersize=10,␣
↪linestyle='', label=str(marker))
>>> plt.legend()
>>> kwplot.show_if_requested()
```

kwplot.**draw_boxes**(*boxes*, *alpha=None*, *color='blue'*, *labels=None*, *centers=False*, *fill=False*, *ax=None*, *lw=2*)

Parameters

- **boxes** (*kwimage.Boxes*)
- **labels** (*List[str]*) – of labels
- **alpha** (*List[float]*) – alpha for each box
- **centers** (*bool*) – draw centers or not
- **lw** (*float*) – linewidth

Example

```
>>> import kwimage
>>> bboxes = kwimage.Boxes([[.1, .1, .6, .3], [.3, .5, .5, .6]], 'xywh')
>>> draw_boxes(bboxes)
>>> #kwplot.autompl()
```

kwplot.**draw_boxes_on_image**(*img*, *boxes*, *color='blue'*, *thickness=1*, *box_format=None*, *colorspace='rgb'*)

Draws boxes on an image.

Parameters

- **img** (*ndarray*) – image to copy and draw on
- **boxes** (*nh.util.Boxes*) – boxes to draw
- **colorspace** (*str*) – string code of the input image colorspace

Example

```
>>> import kwimage
>>> import numpy as np
>>> img = np.zeros((10, 10, 3), dtype=np.uint8)
>>> color = 'dodgerblue'
>>> thickness = 1
>>> boxes = kwimage.Boxes([[1, 1, 8, 8]], 'ltrb')
>>> img2 = draw_boxes_on_image(img, boxes, color, thickness)
>>> assert tuple(img2[1, 1]) == (30, 144, 255)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()  # xdoc: +SKIP
>>> kwplot.figure(doclf=True, fnum=1)
>>> kwplot.imshow(img2)
```

kwplot.**draw_clf_on_image**(*im*, *classes*, *tcx=None*, *probs=None*, *pcx=None*, *border=1*)

Draws classification label on an image.

Works best with image chips sized between 200x200 and 500x500

Parameters

- **im** (*ndarray*) – the image
- **classes** (*Sequence | CategoryTree*) – list of class names
- **tcx** (*int, default=None*) – true class index if known
- **probs** (*ndarray*) – predicted class probs for each class
- **pcx** (*int, default=None*) – predicted class index. (if None but probs is specified uses argmax of probs)

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torch
>>> import kwarray
>>> import kwimage
>>> rng = kwarray.ensure_rng(0)
>>> im = (rng.rand(300, 300) * 255).astype(np.uint8)
>>> classes = ['cls_a', 'cls_b', 'cls_c']
>>> tcx = 1
>>> probs = rng.rand(len(classes))
>>> probs[tcx] = 0
>>> probs = torch.FloatTensor(probs).softmax(dim=0).numpy()
>>> im1_ = kwimage.draw_clf_on_image(im, classes, tcx, probs)
>>> probs[tcx] = .9
>>> probs = torch.FloatTensor(probs).softmax(dim=0).numpy()
>>> im2_ = kwimage.draw_clf_on_image(im, classes, tcx, probs)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(im1_, colorspace='rgb', pnum=(1, 2, 1), fnum=1, doclf=True)
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.imshow(im2_, colorspace='rgb', pnum=(1, 2, 2), fnum=1)
>>> kwplot.show_if_requested()
```

kwplot.**draw_line_segments**(*pts1*, *pts2*, *ax=None*, ***kwargs*)

draws *N* line segments between *N* pairs of points

Parameters

- **pts1** (*ndarray*) – Nx2
- **pts2** (*ndarray*) – Nx2
- **ax** (*None*) – (default = None)
- ****kwargs** – lw, alpha, colors

Example

```
>>> import numpy as np
>>> import kwplot
>>> pts1 = np.array([(.1, .8), (.6, .8)])
>>> pts2 = np.array([(.6, .7), (.4, .1)])
>>> kwplot.figure(fnum=None)
>>> draw_line_segments(pts1, pts2)
>>> # xdoc: +REQUIRES(--show)
>>> import matplotlib.pyplot as plt
>>> ax = plt.gca()
>>> ax.set_xlim(0, 1)
>>> ax.set_ylim(0, 1)
>>> kwplot.show_if_requested()
```

kwplot.**draw_points**(*xy*, *color='blue'*, *class_idxs=None*, *classes=None*, *ax=None*, *alpha=None*, *radius=1*, ***kwargs*)

Parameters

xy (*ndarray*) – of points.

Example

```
>>> from kwplot.mpl_draw import *  # NOQA
>>> import kwimage
>>> xy = kwimage.Points.random(10).xy
>>> draw_points(xy, radius=0.01)
>>> draw_points(xy, class_idxs=np.random.randint(0, 3, 10),
>>>         radius=0.01, classes=['a', 'b', 'c'], color='classes')
```

kwplot.**draw_text_on_image**(*img*, *text*, *org=None*, *return_info=False*, ***kwargs*)

Draws multiline text on an image using opencv

Parameters

- **img** (*ndarray | None | dict*) – Generally a numpy image to draw on (inplace). Otherwise a canvas will be constructed such that the text will fit. The user may specify a dictionary with keys width and height to have more control over the constructed canvas.

- **text** (*str*) – text to draw
- **org** (*Tuple[int, int]*) – The x, y location of the text string "anchor" in the image as specified by halign and valign. For instance, If valign='bottom', halign='left', this is the bottom left corner.
- **return_info** (*bool, default=False*) – if True, also returns information about the positions the text was drawn on.
- ****kwargs** – color (tuple): default blue thickness (int): defaults to 2 fontFace (int): defaults to cv2.FONT_HERSHEY_SIMPLEX fontScale (float): defaults to 1.0 valign (str, default='bottom'):

  either top, center, or bottom. NOTE: this default may change to "top" in the future.

  halign (str, default='left'):
  either left, center, or right

  border (dict | int):
  If specified as an integer, draws a black border with that given thickness. If specified as a dictionary, draws a border with color specified parameters.

  "color": border color, defaults to "black". "thickness": border thickness, defaults to 1.

Returns
the image that was drawn on

Return type
ndarray

Note: The image is modified inplace. If the image is non-contiguous then this returns a UMat instead of a ndarray, so be carefull with that.

References

https://stackoverflow.com/questions/27647424/ https://stackoverflow.com/questions/51285616/opencvs-gettextsize-and-puttext-return-wrong-size-and-chop-letters-with-low

Example

```
>>> import kwimage
>>> img = kwimage.grab_test_image(space='rgb')
>>> img2 = kwimage.draw_text_on_image(img.copy(), 'FOOBAR', org=(0, 0), valign='top
↪')
>>> assert img2.shape == img.shape
>>> assert np.any(img2 != img)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img2)
>>> kwplot.show_if_requested()
```

Example

```
>>> import kwimage
>>> # Test valign
>>> img = kwimage.grab_test_image(space='rgb', dsize=(500, 500))
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(0, 0),␣
↪valign='top', border=2)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(150, 0),␣
↪valign='center', border=2)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(300, 0),␣
↪valign='bottom', border=2)
>>> # Test halign
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(250, 100),␣
↪halign='right', border=2)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(250, 250),␣
↪halign='center', border=2)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(250, 400),␣
↪halign='left', border=2)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img2)
>>> kwplot.show_if_requested()
```

Example

```
>>> # Ensure the function works with float01 or uint255 images
>>> import kwimage
>>> img = kwimage.grab_test_image(space='rgb')
>>> img = kwimage.ensure_float01(img)
>>> img2 = kwimage.draw_text_on_image(img, 'FOOBAR\nbazbiz\nspam', org=(0, 0),␣
↪valign='top', border=2)
```

Example

```
>>> # Test dictionary border
>>> import kwimage
>>> img = kwimage.draw_text_on_image(None, 'hello\neveryone', org=(100, 100),␣
↪valign='top', halign='center', border={'color': 'green', 'thickness': 9})
>>> #img = kwimage.draw_text_on_image(None, 'hello\neveryone', org=(0, 0), valign='top
↪')
>>> #img = kwimage.draw_text_on_image(None, 'hello', org=(0, 60), valign='top', halign=
↪'center', border=0)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img)
>>> kwplot.show_if_requested()
```

Example

```
>>> # Test dictionary image
>>> import kwimage
>>> img = kwimage.draw_text_on_image({'width': 300}, 'good\nPropogate', org=(150,␣
↪0), valign='top', halign='center', border={'color': 'green', 'thickness': 0})
>>> print('img.shape = {!r}'.format(img.shape))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img)
>>> kwplot.show_if_requested()
```

Example

```
>>> import ubelt as ub
>>> import kwimage
>>> grid = list(ub.named_product({
>>>     'halign': ['left', 'center', 'right', None],
>>>     'valign': ['top', 'center', 'bottom', None],
>>>     'border': [0, 3]
>>> }))
>>> canvases = []
>>> text = 'small-line\na-much-much-much-bigger-line\nanother-small\n.'
>>> for kw in grid:
>>>     header = kwimage.draw_text_on_image({}, ub.repr2(kw, compact=1), color='blue
↪')
>>>     canvas = kwimage.draw_text_on_image({'color': 'white'}, text, org=None,␣
↪**kw)
>>>     canvases.append(kwimage.stack_images([header, canvas], axis=0, bg_
↪value=(255, 255, 255), pad=5))
>>> # xdoc: +REQUIRES(--show)
>>> canvas = kwimage.stack_images_grid(canvases, pad=10, bg_value=(255, 255, 255))
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(canvas)
>>> kwplot.show_if_requested()
```

kwplot.**ensure_fnum**(*fnum*)

kwplot.**figure**(*fnum=None*, *pnum=(1, 1, 1)*, *title=None*, *figtitle=None*, *doclf=False*, *docla=False*, *projection=None*, ***kwargs*)

http://matplotlib.org/users/gridspec.html

Parameters

- **fnum** (*int*) – fignum = figure number
- **pnum** (*int, str, or tuple(int, int, int)*) – plotnum = plot tuple
- **title** (*str*) – (default = None)
- **figtitle** (*None*) – (default = None)
- **docla** (*bool*) – (default = False)

- **doclf** (*bool*) – (default = False)

Returns
fig

Return type
mpl.figure.Figure

Example

```
>>> import kwplot
>>> kwplot.autompl()
>>> import matplotlib.pyplot as plt
>>> fnum = 1
>>> fig = figure(fnum, (2, 2, 1))
>>> fig.gca().text(0.5, 0.5, "ax1", va="center", ha="center")
>>> fig = figure(fnum, (2, 2, 2))
>>> fig.gca().text(0.5, 0.5, "ax2", va="center", ha="center")
>>> show_if_requested()
```

Example

```
>>> import kwplot
>>> kwplot.autompl()
>>> import matplotlib.pyplot as plt
>>> fnum = 1
>>> fig = figure(fnum, (2, 2, 1))
>>> fig.gca().text(0.5, 0.5, "ax1", va="center", ha="center")
>>> fig = figure(fnum, (2, 2, 2))
>>> fig.gca().text(0.5, 0.5, "ax2", va="center", ha="center")
>>> fig = figure(fnum, (2, 4, (1, slice(1, None))))
>>> fig.gca().text(0.5, 0.5, "ax3", va="center", ha="center")
>>> show_if_requested()
```

kwplot.**imshow**(*img*, *fnum=None*, *pnum=None*, *xlabel=None*, *title=None*, *figtitle=None*, *ax=None*, *norm=None*, *cmap=None*, *data_colorbar=False*, *colorspace='rgb'*, *interpolation='nearest'*, *alpha=None*, *show_ticks=False*, ***kwargs*)

Parameters

- **img** (*ndarray*) – image data. Height, Width, and Channel dimensions can either be in standard (H, W, C) format or in (C, H, W) format. If C in [3, 4], we assume data is in the rgb / rgba colorspace by default.
- **colorspace** (*str*) – if the data is 3-4 channels, this indicates the colorspace 1 channel data is assumed grayscale. 4 channels assumes alpha.
- **interpolation** (*str*) – either nearest (default), bicubic, bilinear
- **norm** (*bool*) – if True, normalizes the image intensities to fit in a colormap.
- **cmap** (*mpl.colors.Colormap | None*) – color map used if data is not starndard image data
- **data_colorbar** (*bool*) – if True, displays a color scale indicating how colors map to image intensities.

- **fnum** (*int | None*) – figure number
- **pnum** (*tuple | None*) – plot number
- **xlabel** (*str | None*) – sets the label for the x axis
- **title** (*str | None*) – set axes title (if ax is not given)
- **figtitle** (*str | None*) – set figure title (if ax is not given)
- **ax** (*mpl.axes.Axes | None*) – axes to draw on (alternative to fnum and pnum)
- ****kwargs** – docla, doclf, projection

Returns
(fig, ax)

Return type
tuple

kwplot.**legend**(*loc='best'*, *fontproperties=None*, *size=None*, *fc='w'*, *alpha=1*, *ax=None*, *handles=None*)

Parameters

- **loc** (*str*) – (default = 'best') one of 'best', 'upper right', 'upper left', 'lower left', 'lower right', 'right', 'center left', 'center right', 'lower center', or 'upper center'.
- **fontproperties** (*None*) – (default = None)
- **size** (*None*) – (default = None)

kwplot.**make_conv_images**(*conv*, *color=None*, *norm_per_feat=True*)

Convert convolutional weights to a list of visualize-able images

Parameters

- **conv** (*torch.nn.Conv2d*) – a torch convolutional layer
- **color** (*bool*) – if True output images are colorized
- **norm_per_feat** (*bool*) – if True normalizes over each feature separately, otherwise normalizes all features together.

Return type
ndarray

Todo:

- [ ] better normalization options

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> conv = torch.nn.Conv2d(3, 9, (5, 7))
>>> weights_tohack = conv.weight[0:7].data.numpy()
>>> weights_flat = make_conv_images(conv, norm_per_feat=False)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwimage
>>> import kwplot
>>> stacked = kwimage.stack_images_grid(weights_flat, chunksize=5, overlap=-1)
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.imshow(stacked)
>>> kwplot.show_if_requested()
```

kwplot.**make_heatmask**(*probs*, *cmap='plasma'*, *with_alpha=1.0*, *space='rgb'*, *dsize=None*)

Colorizes a single-channel intensity mask (with an alpha channel)

Parameters

- **probs** (*ndarray*) – 2D probability map with values between 0 and 1
- **cmap** (*str*) – mpl colormap
- **with_alpha** (*float*) – between 0 and 1, uses probs as the alpha multipled by this number.
- **space** (*str*) – output colorspace
- **dsize** (*tuple*) – if not None, then output is resized to W,H=dsize

SeeAlso:

kwimage.overlay_alpha_images

Example

```
>>> # xdoc: +REQUIRES(module:matplotlib)
>>> from kwimage.im_draw import *  # NOQA
>>> probs = np.tile(np.linspace(0, 1, 10), (10, 1))
>>> heatmask = make_heatmask(probs, with_alpha=0.8, dsize=(100, 100))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.imshow(heatmask, fnum=1, doclf=True, colorspace='rgb')
>>> kwplot.show_if_requested()
```

kwplot.**make_legend_img**(*label_to_color*, *dpi=96*, *shape=(200, 200)*, *mode='line'*, *transparent=False*)

Makes an image of a categorical legend

Parameters

label_to_color (*Dict[str, kwimage.Color]*) – mapping from string label to the color.

CommandLine

```
xdoctest -m kwplot.mpl_make make_legend_img --show
```

Example

```
>>> # xdoctest: +REQUIRES(module:kwplot)
>>> import kwplot
>>> import kwimage
>>> label_to_color = {
>>>     'blue': kwimage.Color('blue').as01(),
>>>     'red': kwimage.Color('red').as01(),
>>>     'green': 'green',
```

(continues on next page)

(continued from previous page)

```
>>>     'yellow': 'yellow',
>>>     'orangered': 'orangered',
>>> }
>>> img = make_legend_img(label_to_color, transparent=True)
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.autompl()
>>> kwplot.imshow(img)
>>> kwplot.show_if_requested()
```

kwplot.**make_orimask**(*radians*, *mag=None*, *alpha=1.0*)

Makes a colormap in HSV space where the orientation changes color and mag changes the saturation/value.

Parameters

- **radians** (*ndarray*) – orientation in radians
- **mag** (*ndarray*) – magnitude (must be normalized between 0 and 1)
- **alpha** (*float | ndarray*) – if False or None, then the image is returned without alpha if a float, then mag is scaled by this and used as the alpha channel if an ndarray, then this is explicilty set as the alpha channel

Returns

an rgb / rgba image in 01 space

Return type

ndarray[float32]

SeeAlso:

kwimage.overlay_alpha_images

Example

```
>>> # xdoc: +REQUIRES(module:matplotlib)
>>> from kwimage.im_draw import *  # NOQA
>>> x, y = np.meshgrid(np.arange(64), np.arange(64))
>>> dx, dy = x - 32, y - 32
>>> radians = np.arctan2(dx, dy)
>>> mag = np.sqrt(dx ** 2 + dy ** 2)
>>> orimask = make_orimask(radians, mag)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.imshow(orimask, fnum=1, doclf=True, colorspace='rgb')
>>> kwplot.show_if_requested()
```

kwplot.**make_vector_field**(*dx*, *dy*, *stride=0.02*, *thresh=0.0*, *scale=1.0*, *alpha=1.0*, *color='strawberry'*, *thickness=1*, *tipLength=0.1*, *line_type='aa'*)

Create an image representing a 2D vector field.

Parameters

- **dx** (*ndarray*) – grid of vector x components
- **dy** (*ndarray*) – grid of vector y components

- **stride** (*int | float*) – sparsity of vectors, int specifies stride step in pixels, a float specifies it as a percentage.
- **thresh** (*float*) – only plot vectors with magnitude greater than thres
- **scale** (*float*) – multiply magnitude for easier visualization
- **alpha** (*float*) – alpha value for vectors. Non-vector regions receive 0 alpha (if False, no alpha channel is used)
- **color** (*str | tuple | kwimage.Color*) – RGB color of the vectors
- **thickness** (*int, default=1*) – thickness of arrows
- **tipLength** (*float, default=0.1*) – fraction of line length
- **line_type** (*int*) – either cv2.LINE_4, cv2.LINE_8, or cv2.LINE_AA

Returns
vec_img: an rgb/rgba image in 0-1 space

Return type
ndarray[float32]

SeeAlso:
kwimage.overlay_alpha_images

DEPRECATED USE: draw_vector_field instead

Example

```
>>> x, y = np.meshgrid(np.arange(512), np.arange(512))
>>> dx, dy = x - 256.01, y - 256.01
>>> radians = np.arctan2(dx, dy)
>>> mag = np.sqrt(dx ** 2 + dy ** 2)
>>> dx, dy = dx / mag, dy / mag
>>> img = make_vector_field(dx, dy, scale=10, alpha=False)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img)
>>> kwplot.show_if_requested()
```

kwplot.**multi_plot**(*xdata=None*, *ydata=None*, *xydata=None*, ***kwargs*)

plots multiple lines, bars, etc…

One function call that concisely describes the all of the most commonly used parameters needed when plotting a bar / line char. This is especially useful when multiple plots are needed in the same domain.

Parameters

- **xdata** (*List[ndarray] | Dict[str, ndarray] | ndarray*) – x-coordinate data common to all y-coordinate values or xdata for each line/bar in ydata. Mutually exclusive with xydata.
- **ydata** (*List[ndarray] | Dict[str, ndarray] | ndarray*) – y-coordinate values for each line/bar to plot. Can also be just a single ndarray of scalar values. Mutually exclusive with xydata.
- **xydata** (*Dict[str, Tuple[ndarray, ndarray]]*) – mapping from labels to a tuple of xdata and ydata for a each line.

- ****kwargs** –

  fnum (int):
  figure number to draw on

  pnum (Tuple[int, int, int]):
  plot number to draw on within the figure, e.g. (1, 1, 1)

  label (List|Dict):
  if you specified ydata as a List[ndarray] this is the label for each line in that list. Note this is unnecessary if you specify input as a dictionary mapping labels to lines.

  color (str|List|Dict):
  either a special color code, a single color, or a color for each item in ydata. In the later case, this should be specified as either a list or a dict depending on how ydata was specified.

  marker (str|List|Dict):
  type of matplotlib marker to use at every data point. Can be specified for all lines jointly or for each line independently.

  transpose (bool, default=False):
  swaps x and y data.

  kind (str, default='plot'):
  The kind of plot. Can either be 'plot' or 'bar'. We parse these other kwargs if:

  if kind='plot':
  spread

  if kind='bar':
  stacked, width

  Misc:
  use_legend (bool): … legend_loc (str):

  one of 'best', 'upper right', 'upper left', 'lower left', 'lower right', 'right', 'center left', 'center right', 'lower center', or 'upper center'.

  Layout:
  xlabel (str): label for x-axis ylabel (str): label for y-axis title (str): title for the axes figtitle (str): title for the figure

  xscale (str): can be one of [linear, log, logit, symlog] yscale (str): can be one of [linear, log, logit, symlog]

  xlim (Tuple[float, float]): low and high x-limit of axes ylim (Tuple[float, float]): low and high y-limit of axes xmin (float): low x-limit of axes, mutex with xlim xmax (float): high x-limit of axes, mutex with xlim ymin (float): low y-limit of axes, mutex with ylim ymax (float): high y-limit of axes, mutex with ylim

  titlesize (float): … legendsize (float): … labelsize (float): …

  Grid:
  gridlinewidth (float): … gridlinestyle (str): …

  Ticks:
  num_xticks (int): number of x ticks num_yticks (int): number of y ticks tickwidth (float): … ticklength (float): … ticksize (float): … xticklabels (list): list of x-tick labels, overrides num_xticks yticklabels (list): list of y-tick labels, overrides num_yticks xtick_rotation (float): xtick rotation in degrees ytick_rotation (float): yt-ick rotation in degrees

Data:

spread (List | Dict): Plots a spread around plot lines usually
indicating standard deviation

markersize (float|List|Dict): marker size for all or each plot markeredgewidth (float|List|Dict): marker edge width for all or each plot linewidth (float|List|Dict): line width for all or each plot linestyle (str|List|Dict): line style for all or each plot

---

Note: any plot_kw key can be a scalar (corresponding to all ydatas), a list if ydata was specified as a list, or a dict if ydata was specified as a dict.

plot_kw_keys = ['label', 'color', 'marker', 'markersize',
'markeredgewidth', 'linewidth', 'linestyle']

Note: In general this should be deprecated in favor of using seaborn

---

Returns
ax : the axes that was drawn on

Return type
matplotlib.axes.Axes

References

matplotlib.org/examples/api/barchart_demo.html

Example

```
>>> import kwplot
>>> kwplot.autompl()
>>> # The new way to use multi_plot is to pass ydata as a dict of lists
>>> ydata = {
>>>     'spam': [1, 1, 2, 3, 5, 8, 13],
>>>     'eggs': [3, 3, 3, 3, 3, np.nan, np.nan],
>>>     'jamμ': [5, 3, np.nan, 1, 2, np.nan, np.nan],
>>>     'pram': [4, 2, np.nan, 0, 0, np.nan, 1],
>>> }
>>> ax = kwplot.multi_plot(ydata=ydata, title='μμμ',
>>>                        xlabel='\nfdsμμμ', linestyle='--')
>>> kwplot.show_if_requested()
```

Example

```
>>> # Old way to use multi_plot is a list of lists
>>> import kwplot
>>> kwplot.autompl()
>>> xdata = [1, 2, 3, 4, 5]
>>> ydata_list = [[1, 2, 3, 4, 5], [3, 3, 3, 3, 3], [5, 4, np.nan, 2, 1], [4, 3, np.
↪nan, 1, 0]]
>>> kwargs = {'label': ['spam', 'eggs', 'jamμ', 'pram'],  'linestyle': '-'}
>>> #ax = multi_plot(xdata, ydata_list, title='$\phi_1(\\vec{x})$', xlabel='\nfds',␣
↪**kwargs)
>>> ax = multi_plot(xdata, ydata_list, title='μμμ', xlabel='\nfdsμμμ', **kwargs)
>>> kwplot.show_if_requested()
```

Example

```
>>> # Simple way to use multi_plot is to pass xdata and ydata exactly
>>> # like you would use plt.plot
>>> import kwplot
>>> kwplot.autompl()
>>> ax = multi_plot([1, 2, 3], [4, 5, 6], fnum=4, label='foo')
>>> kwplot.show_if_requested()
```

Example

```
>>> import kwplot
>>> kwplot.autompl()
>>> xydata = {'a': ([0, 1, 2], [0, 1, 2]), 'b': ([0, 2, 4], [2, 1, 0])}
>>> ax = kwplot.multi_plot(xydata=xydata, fnum=4)
>>> kwplot.show_if_requested()
```

Example

```
>>> import kwplot
>>> kwplot.autompl()
>>> ydata = {'a': [0, 1, 2], 'b': [1, 2, 1], 'c': [4, 4, 4, 3, 2]}
>>> kwargs = {
>>>     'spread': {'a': [.2, .3, .1], 'b': .2},
>>>     'xlim': (-1, 5),
>>>     'xticklabels': ['foo', 'bar'],
>>>     'xtick_rotation': 90,
>>> }
>>> ax = kwplot.multi_plot(ydata=ydata, fnum=4, **kwargs)
>>> kwplot.show_if_requested()
```

kwplot.**next_fnum**(*new_base=None*)

kwplot.**phantom_legend**(*label_to_color*, *mode='line'*, *ax=None*, *legend_id=None*, *loc=0*)

Creates a legend on an axis based on a label-to-color map.

Parameters
label_to_color (*Dict[str, kwimage.Color]*) – mapping from string label to the color.

Todo:

- [ ] More docs and ensure this exists in the right place

---

kwplot.**plot_convolutional_features**(*conv*, *limit=144*, *colorspace='rgb'*, *fnum=None*, *nCols=None*, *voxels=False*, *alpha=0.2*, *labels=False*, *normaxis=None*, *_hack_2drows=False*)

Plots the convolutional layers to a matplotlib pyplot.

The convolutional filters (kernels) are stored into a grid and saved to disk as a Maplotlib figure. The convolutional filters, if it has one channel, will be stored as an intensity imgage. If a colorspace is specified and there are three input channels, the convolutional filters will be represented as an RGB image.

In the event that 2 or 4+ filters are displayed, the different channels will be flattened and showed as distinct outputs in the grid.

Todo:

- [ ] refactor to use make_conv_images

Parameters

- **conv** (*torch.nn.modules.conv._ConvNd*) – torch convolutional layer with weights to draw
- **limit** (*int*) – the limit on the number of filters drawn in the figure, achieved by simply dropping any filters past the limit starting at the first filter. Detaults to 144.
- **colorspace** (*str*) – the colorspace seen by the convolutional filter (if applicable), so we can convert to rgb for display.
- **voxels** (*bool*) – if True, and we have a 3d conv, show the voxels
- **alpha** (*float*) – only applicable if voxels=True
- **stride** (*list*) – only applicable if voxels=True

Returns
fig - a Matplotlib figure

Return type
matplotlib.figure.Figure

References

https://matplotlib.org/devdocs/gallery/mplot3d/voxels.html

Example

```
>>> # xdoctest: +REQUIRES(module:torch)
>>> conv = torch.nn.Conv2d(3, 9, (5, 7))
>>> plot_convolutional_features(conv, colorspace=None, fnum=None, limit=2)
```

Example

```
>>> # xdoctest: +REQUIRES(--comprehensive)
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torchvision
>>> # 2d uncolored gray-images
>>> conv = torch.nn.Conv3d(1, 2, (3, 4, 5))
>>> plot_convolutional_features(conv, colorspace=None, fnum=1, limit=2)
```

```
>>> # 2d colored rgb-images
>>> conv = torch.nn.Conv3d(3, 2, (6, 4, 5))
>>> plot_convolutional_features(conv, colorspace='rgb', fnum=1, limit=2)
```

```
>>> # 2d uncolored rgb-images
>>> conv = torch.nn.Conv3d(3, 2, (6, 4, 5))
>>> plot_convolutional_features(conv, colorspace=None, fnum=1, limit=2)
```

```
>>> # 3d gray voxels
>>> conv = torch.nn.Conv3d(1, 2, (6, 4, 5))
>>> plot_convolutional_features(conv, colorspace=None, fnum=1, voxels=True,
>>>                             limit=2)
```

```
>>> # 3d color voxels
>>> conv = torch.nn.Conv3d(3, 2, (6, 4, 5))
>>> plot_convolutional_features(conv, colorspace='rgb', fnum=1,
>>>                             voxels=True, alpha=1, limit=3)
```

```
>>> # hack the nice resnet weights into 3d-space
>>> # xdoctest: +REQUIRES(--network)
>>> import torchvision
>>> model = torchvision.models.resnet50(pretrained=True)
>>> conv = torch.nn.Conv3d(3, 1, (7, 7, 7))
>>> weights_tohack = model.conv1.weight[0:7].data.numpy()
>>> # normalize each weight for nice colors, then place in the conv3d
>>> for w in weights_tohack:
...     w[:] = (w - w.min()) / (w.max() - w.min())
>>> weights_hacked = weights_tohack.transpose(1, 0, 2, 3)[None, :]
>>> conv.weight.data[:] = torch.FloatTensor(weights_hacked)
```

```
>>> plot_convolutional_features(conv, colorspace='rgb', fnum=1, voxels=True, alpha=.
↪6)
```

```
>>> plot_convolutional_features(conv, colorspace='rgb', fnum=2, voxels=False,␣
↪alpha=.9)
```

Example

```
>>> # xdoctest: +REQUIRES(--network)
>>> # xdoctest: +REQUIRES(module:torch)
>>> import torchvision
>>> model = torchvision.models.resnet50(pretrained=True)
>>> conv = model.conv1
>>> plot_convolutional_features(conv, colorspace='rgb', fnum=None)
```

kwplot.**plot_matrix**(*matrix*, *index=None*, *columns=None*, *rot=90*, *ax=None*, *grid=True*, *label=None*, *zerodiag=False*, *cmap='viridis'*, *showvals=False*, *showzero=True*, *logscale=False*, *xlabel=None*, *ylabel=None*, *fnum=None*, *pnum=None*)

Helper for plotting confusion matrices

Parameters

matrix (*ndarray | pd.DataFrame*) – if a data frame then index, columns, xlabel, and ylabel will be defaulted to sensible values.

Todo:

- [ ] Finish args docs
- [ ] Replace internals with seaborn

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwplot.mpl_draw import *  # NOQA
>>> import pandas as pd
>>> classes = ['cls1', 'cls2', 'cls3']
>>> matrix = np.array([[2, 2, 1], [3, 1, 0], [1, 0, 0]])
>>> matrix = pd.DataFrame(matrix, index=classes, columns=classes)
>>> matrix.index.name = 'real'
>>> matrix.columns.name = 'pred'
>>> plot_matrix(matrix, showvals=True)
>>> # xdoc: +REQUIRES(--show)
>>> import matplotlib.pyplot as plt
>>> import kwplot
>>> kwplot.autompl()
>>> plot_matrix(matrix, showvals=True)
```

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwplot.mpl_draw import *  # NOQA
>>> matrix = np.array([[2, 2, 1], [3, 1, 0], [1, 0, 0]])
>>> plot_matrix(matrix)
>>> # xdoc: +REQUIRES(--show)
>>> import matplotlib.pyplot as plt
>>> import kwplot
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.autompl()
>>> plot_matrix(matrix)
```

Example

```
>>> # xdoctest: +REQUIRES(module:pandas)
>>> from kwplot.mpl_draw import *  # NOQA
>>> matrix = np.array([[2, 2, 1], [3, 1, 0], [1, 0, 0]])
>>> classes = ['cls1', 'cls2', 'cls3']
>>> plot_matrix(matrix, index=classes, columns=classes)
```

kwplot.**plot_surface3d**(*xgrid*, *ygrid*, *zdata*, *xlabel=None*, *ylabel=None*, *zlabel=None*, *wire=False*, *mode=None*, *contour=False*, *rstride=1*, *cstride=1*, *pnum=None*, *labelkw=None*, *xlabelkw=None*, *ylabelkw=None*, *zlabelkw=None*, *titlekw=None*, **args*, ***kwargs*)

References

http://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html

Example

```
>>> # DISABLE_DOCTEST
>>> import kwplot
>>> import matplotlib as mpl
>>> import kwimage
>>> shape=(19, 19)
>>> sigma1, sigma2 = 2.0, 1.0
>>> ybasis = np.arange(shape[0])
>>> xbasis = np.arange(shape[1])
>>> xgrid, ygrid = np.meshgrid(xbasis, ybasis)
>>> sigma = [sigma1, sigma2]
>>> gausspatch = kwimage.gaussian_patch(shape, sigma=sigma)
>>> title = 'ksize={!r}, sigma={!r}'.format(shape, (sigma1, sigma2))
>>> kwplot.plot_surface3d(xgrid, ygrid, gausspatch, rstride=1, cstride=1,
>>>                       cmap=mpl.cm.coolwarm, title=title)
>>> kwplot.show_if_requested()
```

kwplot.**render_figure_to_image**(*fig*, *dpi=None*, *transparent=None*, ***savekw*)

Saves a figure as an image in memory.

Parameters

- **fig** (*matplotlib.figure.Figure*) – figure to save
- **dpi** (*Optional[int | str]*) – The resolution in dots per inch. If *None* it will default to the value `savefig.dpi` in the matplotlibrc file. If 'figure' it will set the dpi to be the value of the figure.
- **transparent** (*bool*) – If *True*, the axes patches will all be transparent; the figure patch will also be transparent unless facecolor and/or edgecolor are specified via kwargs.

- ****savekw** – other keywords passed to `fig.savefig`. Valid keywords include: facecolor, edgecolor, orientation, papertype, format, pad_inches, frameon.

Returns
an image in RGB or RGBA format.

Return type
np.ndarray

Note: Be sure to use *fig.set_size_inches* to an appropriate size before calling this function.

Example

```
>>> import kwplot
>>> fig = kwplot.figure(fnum=1, doclf=True)
>>> ax = fig.gca()
>>> ax.cla()
>>> ax.plot([0, 10], [0, 10])
>>> canvas_rgb = kwplot.render_figure_to_image(fig, transparent=False)
>>> canvas_rgba = kwplot.render_figure_to_image(fig, transparent=True)
>>> assert canvas_rgb.shape[2] == 3
>>> assert canvas_rgba.shape[2] == 4
>>> # xdoctest: +REQUIRES(--show)
>>> kwplot.autompl()
>>> kwplot.imshow(canvas_rgb, fnum=2)
>>> kwplot.show_if_requested()
```

kwplot.**set_figtitle**(*figtitle*, *subtitle=''*, *forcefignum=True*, *incanvas=True*, *size=None*, *fontfamily=None*, *fontweight=None*, *fig=None*)

A wrapper around subtitle that also sets the canvas window title if using a Qt backend.

Parameters

- **figtitle** (*str*)
- **subtitle** (*str*)
- **forcefignum** (*bool*) – (default = True)
- **incanvas** (*bool*) – (default = True)
- **fontfamily** (*None*) – (default = None)
- **fontweight** (*None*) – (default = None)
- **size** (*None*) – (default = None)
- **fig** (*None*) – (default = None)

CommandLine

```
python -m kwplot.mpl_core set_figtitle --show
```

Example

```
>>> # DISABLE_DOCTEST
>>> autompl()
>>> fig = figure(fnum=1, doclf=True)
>>> result = set_figtitle(figtitle='figtitle', fig=fig)
>>> # xdoc: +REQUIRES(--show)
>>> show_if_requested()
```

kwplot.**set_mpl_backend**(*backend*, *verbose=None*)

Parameters
backend (*str*) – name of backend to use (e.g. Agg, PyQt)

kwplot.**show_if_requested**(*N=1*)

Used at the end of tests. Handles command line arguments for saving figures

Referencse:
http://stackoverflow.com/questions/4325733/save-a-subplot-in-matplotlib

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

k

INDEX

L

M

N

P

R

S